

FiCoS: a fine-grained and coarse-grained GPU-powered deterministic simulator for biochemical networks

Andrea Tangherloni, Marco S. Nobile, Paolo Cazzaniga, Giulia Capitoli, Simone Spolaor, Leonardo Rundo, Giancarlo Mauri, and Daniela Besozzi

1 Runge-Kutta methods

In this section we provide the mathematical notation at the basis of numerical integration methods for systems of Ordinary Differential Equations (ODEs). Given a system of n ODEs, its Cauchy problem is defined as:

$$\begin{cases} \frac{d\mathbf{X}}{dt} &= f(t, \mathbf{X}) \\ \mathbf{X}(t_0) &= \mathbf{x}_0, \end{cases} \quad (1)$$

where $\mathbf{X}(t) = (X_1(t), X_2(t), \dots, X_N(t))$ and $t \in [t_0, t_{max}]$.

A generic Runge-Kutta method of order s and stages r is defined as:

$$\mathbf{X}(t_{n+1}) \simeq \mathbf{x}_{n+1} = \mathbf{x}_n + h \sum_{i=1}^s \beta_i k_i, \quad (2)$$

where h is the integration step-size used during the resolution of the ODE system. The auxiliary variables k_i are given by the following relationship:

$$k_i = f(t_n + c_i h, \mathbf{x}_n + h \sum_{j=1}^{s-1} \alpha_{ij} k_j), \text{ with } i = 1, \dots, s. \quad (3)$$

The coefficients α_{ij} , β_i and c_i allow for characterizing every Runge-Kutta method, which can be represented in the so-called Butcher tableau (see Table 1).

Table 1: Butcher tableau of a generic Runge-Kutta method

$$\begin{array}{c|cccc} c_1 & \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1s} \\ c_2 & \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & \alpha_{s1} & \alpha_{s2} & \cdots & \alpha_{ss} \\ \hline & \beta_1 & \beta_2 & \cdots & \beta_s \\ & \beta_1^* & \beta_2^* & \cdots & \beta_s^* \end{array} = \begin{array}{c|c} \mathbf{c} & \mathbf{\Lambda} \\ \hline & \boldsymbol{\beta}^T \end{array}$$

The Runge-Kutta methods are partitioned in three classes based on the coefficients represented in the Butcher tableau. To be more precise, the following rules are applied to classify every Runge-Kutta method:

- if the matrix $\mathbf{\Lambda}$ is lower triangular (i.e., $\alpha_{ij} = 0$ for $j > i$, with $i, j = 1, \dots, s$), then the method is said to be *explicit*;
- if the matrix $\mathbf{\Lambda}$ is lower triangular, including the main diagonal, the method is called *semi-implicit*;

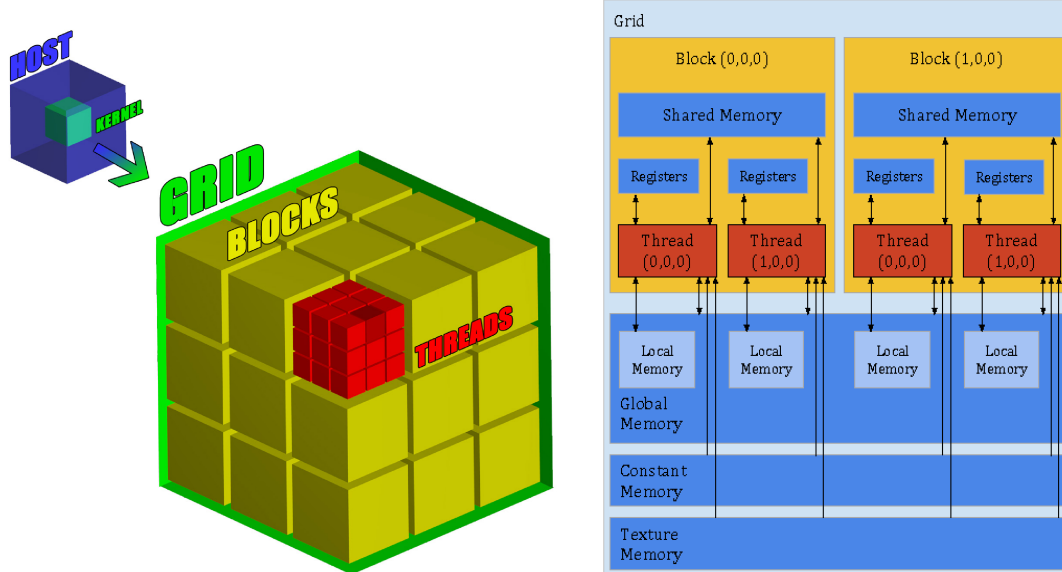


Figure 1: Execution and memory hierarchies of the CUDA's architecture *Left panel* depicts the execution hierarchy, where a single kernel is launched by the CPU (host) and run on the GPU (device) by using multiple threads. When the host launches a kernel, a three-dimensional structure, named grid (green cube) is generated by the device. This grid is partitioned in three-dimensional structures called blocks (yellow cubes), which contain the threads (red cubes). Notice that the number of threads depends on the dimensions of blocks and grids, which must be provided by the programmer. As soon as the grid is created, the device automatically schedules each block on one of the free streaming multiprocessor that are available. Since the GPUs have a different number of streaming multiprocessor, this mechanism allow for a transparent scaling of performances. *Right side* shows the memory hierarchy, which is composed of many different memories with different scopes. Registers and local memories represent the only private memories in which every thread saves its own data. The low access latency shared memory can be used by the threads belonging to the same block to communicate and share information/data. The global memory can be accessed by all the threads as well as by the host, so it is used to the CPU-GPU communications (and vice-versa). Notice that this memory is the largest one, but also the slowest due to its high latency, which was mitigated by means of a cache starting from the Fermi architecture. Texture and constant memories can be assessed only in read by the threads while the host can write the data that are kept constant during the execution of the kernel.

exploit their GPUs for general-purpose computational tasks, following the General-Purpose Computing on Graphics Processing Units (GPGPU) paradigm. Despite the availability of several libraries and high-level Application Programming Interfaces (APIs) provided by CUDA, the direct porting of the CPU code is unfeasible and represents the main challenge of GPGPU computing. As a matter of fact, programmers have to redesign and rewrite the code to fully exploit of the computational power and the massive parallelism of GPUs. In order to develop GPU code, programmers write different kernels (C/C++) that are launched by the CPU (called host) and run on the GPU (called device). Following an execution hierarchy, once a kernel is loaded, the device automatically generates a three-dimensional structures named grid that is divided into different three-dimensional structures called blocks containing the threads, which represent different copies of the kernel (see Fig. 1, left panel). The possible conditional divergences among the threads are managed by CUDA by combining the Single Instruction Multiple Data (SIMD) architecture along with a flexible multi-threading strategy. Besides the execution hierarchy described above, CUDA is based on a memory hierarchy as schematically represented in the right panel of Fig. 1. The memories are partitioned as follows: (i) the global memory, which is accessible from all threads, is the largest one (a few GBs), but at the same time is the slowest one even if the introduction of a cache level starting from the Fermi architecture; (ii) the shared memory is much smaller, up to 112 KB for each streaming multi-processor and limited to 48 KB for each block running over the streaming multi-

processor, but extremely faster with respect the global memory. This memory is used to communicate and share information/data among the threads belonging to the same block; *(iii)* the constant memory, which is cached and read only, is very small (i.e, up to 10 KB for each multi-processor) but can be exploited to store the data that are kept constant during the execution of the kernel; *(iv)* the local memories are represented by private registers and arrays in which the thread saves its own data.

Given these characteristics of the CUDA architecture, the computational power of these many-core devices can be obtained only by optimizing both the thread partitioning and memory usage along with the redesign of the algorithms in a set of appropriate kernels.

3 GPU implementation of FiCoS

In this section, we provide a detailed description of the GPU implementation of FiCoS.

3.1 Data structures and CUDA memory usage

As a first step, starting from the RBM given as input, FiCoS automatically generates the systems of ODEs and encodes both the matrices $\mathbf{H} = (\mathbf{B} - \mathbf{A})^T$ and \mathbf{A} as two arrays of `short2` CUDA vector type, named \mathbf{H}_v and \mathbf{A}_v , respectively. CUDA vector types are multi-dimensional arrays with different components, ranging from 1 to 4, which are addressed by `.x`, `.y`, `.z`, and `.w` components, respectively. Note that we exploited the `short2` CUDA vector type because it is 4-aligned in the memory, meaning that a single instruction is required to fetch a whole entry. We used these data structures to compress and store the matrices \mathbf{A} and \mathbf{H} , which are highly sparse, by removing all zero elements to save memory and at the same time to avoid unnecessary readings from the global memory of the GPU. The following strategy is applied to generate the CUDA vector type \mathbf{H}_v and \mathbf{A}_v :

- for each non-zero element h_{ji} of \mathbf{H} , with $i = 1, \dots, M$ and $j = 1, \dots, N$, we store in the `.x` and `.y` components of \mathbf{H}_v the values i and h_{ji} , respectively;
- for each non-zero element a_{ij} of \mathbf{A} , with $i = 1, \dots, M$ and $j = 1, \dots, N$, we store in the `.x` and `.y` components of \mathbf{A}_v the values j and a_{ij} , respectively.

Besides these two data structures, we use two additional arrays of `int` type, named \mathbf{O}_H and \mathbf{O}_A , to parse \mathbf{H}_v and \mathbf{A}_v inside the GPU. \mathbf{O}_H and \mathbf{O}_A store the offsets to correctly fetch the entries of \mathbf{H}_v and \mathbf{A}_v , respectively. Each thread j , with $j = 0, \dots, N - 1$, reads the j and $j + 1$ elements of \mathbf{O}_H , whose values indicate the first index and the last index (minus 1) to access the \mathbf{H}_v structure. Similarly, \mathbf{O}_A stores the indexes that allow the threads to correctly decode the \mathbf{A}_v structure. Finally, an array \mathbf{K} of type `double` is used to store the values of the kinetic constants. An example of these data structures and their decoding is depicted in Fig. 2.

These CUDA structures allow for improving the performance of FiCoS at two different levels. First, a single instruction is sufficient to either load or store a multi-word vector. This means that the total instruction latency for a particular memory transaction is lower, while the bytes per instruction ratio is higher. Second, by using vector types, a transfer request from a warp has a larger net memory throughput per transaction, yielding a higher bytes per transaction ratio. Therefore, with a lower number of transfer requests, the memory controller can reduce the contentions and achieve a higher overall memory bandwidth utilization.

3.2 Execution workflow

Once the data structures encoding the system of ODEs are generated from the input files (see S2 Text), FiCoS solves the systems of ODEs by means of the DOPRI5 method [2, 3, 4] in the absence of stiffness or the RADAU5 method [5, 6] when the system is stiff. We point out that our implementation was inspired by the source code of Blake Ashby who ported the original `Fortran` code of Hairer and Wanner [5, 4] to `C++`.

Starting from an initial time instant t_0 , the system of ODEs is integrated up to a given maximum simulation time t_{max} . The dynamics of the species appearing in the RBM, described by the ODEs, are sampled and saved at specified time instants within the interval $[t_0, t_{max}]$.

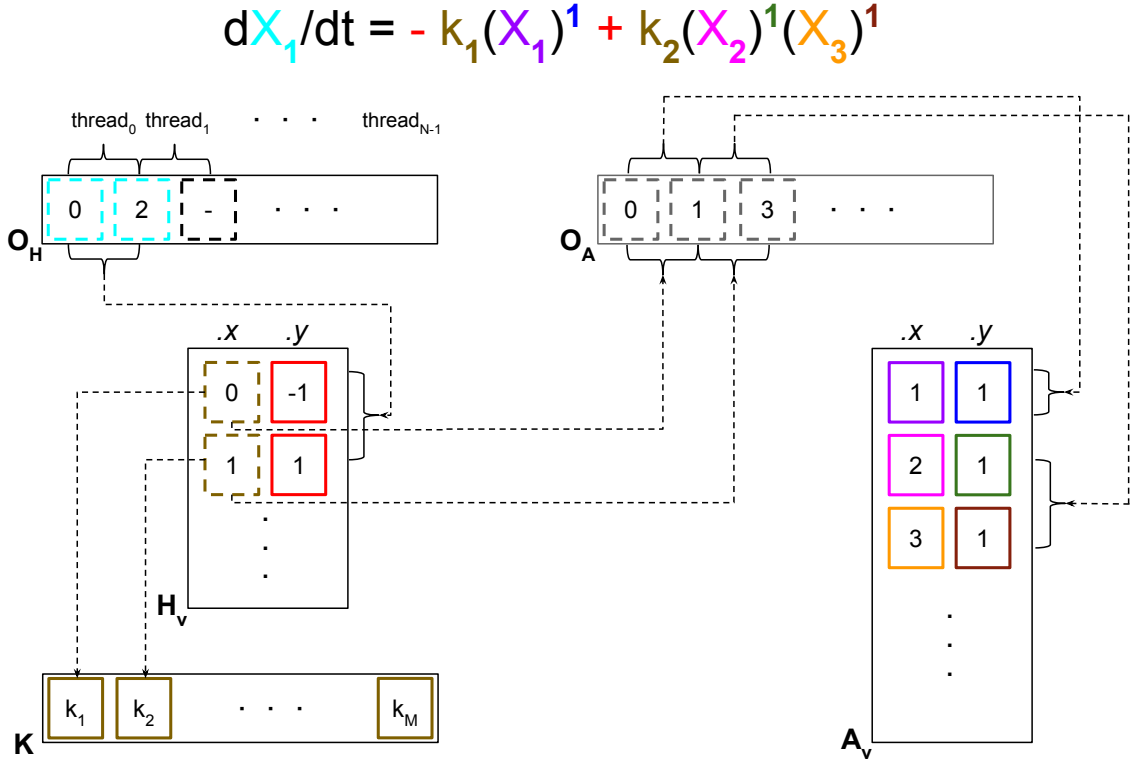


Figure 2: The monomials composing the polynomial function that describes the ODE of the species X_1 , at the top of the figure, are encoded in \mathbf{O}_H , \mathbf{O}_A , \mathbf{H}_v , \mathbf{A}_v and \mathbf{K} . A detailed description of the procedure to decode the data structures is provided below. Note that the ODE is automatically generated using the data structure components with solid borders; the colors used to highlight the terms composing the ODE are also used in the data structure components. Each thread j , with $j = 0, \dots, N - 1$, reads the j and $j + 1$ elements of \mathbf{O}_H (denoted by the lightblue borders). Each thread j fetches the values stored in \mathbf{H}_v , starting from the row indicated by the value of the element j of \mathbf{O}_H up to the row indicated by the value (minus 1) of the element $j + 1$ of \mathbf{O}_H . In this example, thread 0 reads the $.x$ and $.y$ components of the first two rows (i.e., rows 0 and 1) of \mathbf{H}_v . The $.x$ component of each row of \mathbf{H}_v (denoted by dark yellow) indicates both the position of the vector \mathbf{K} —which contains the values of the kinetic constants—and the first index of \mathbf{O}_A where each thread j must read; the $.y$ component (red borders) codifies both the sign and coefficient of the monomial. In this example, the $.x$ and $.y$ components of \mathbf{H}_v codify the coefficients $-1k_1$ and $+1k_2$ of the first and the second term of the ODE, respectively. Afterwards, each thread j fetches the values encoded in \mathbf{O}_A starting from the value stored in the position indicated by the index previously read up to value (minus 1) stored in the next position. The values stored in the $.x$ (purple, magenta, and orange borders) components of \mathbf{A}_v are the indexes of the species, while the $.y$ (blue, dark green, and dark red borders) components codify the stoichiometric coefficients. In this example, the row 0 of \mathbf{A}_v contains the factor $(X_1)^1$, while the rows 1 and 2 allow the thread 0 to generate the factor $(X_2)^1(X_3)^1$. By so doing, the thread 0 is capable of reproducing the polynomial composing the ODE: $-k_1(X_1)^1 + k_2(X_2)^1(X_3)^1$.

FiCoS's workflow can be summarized in 5 distinct phases, as depicted in Fig. 3. Note that only phases P_1 and P_5 are executed by the host (light orange boxes in Fig. 3), while the others are executed by the device (light green boxes in Fig. 3). Phases P_2 , P_3 and P_4 correspond to three different kernels (K_1 , K_2 , and K_3) developed to take advantage of the coarse-grained parallelization strategy. Moreover, these phases recall other lightweight kernels, developed to fully leverage the parallel architecture of the modern GPU, which exploit the dynamic parallelism (DP) [11] to realize the fine-grained parallelization strategy of the aforementioned numerical integration methods. We describe hereafter each phase with its CUDA kernels to explain our novel parallelization strategy that allows FiCoS to achieve the relevant speed-up shown in the Results section.

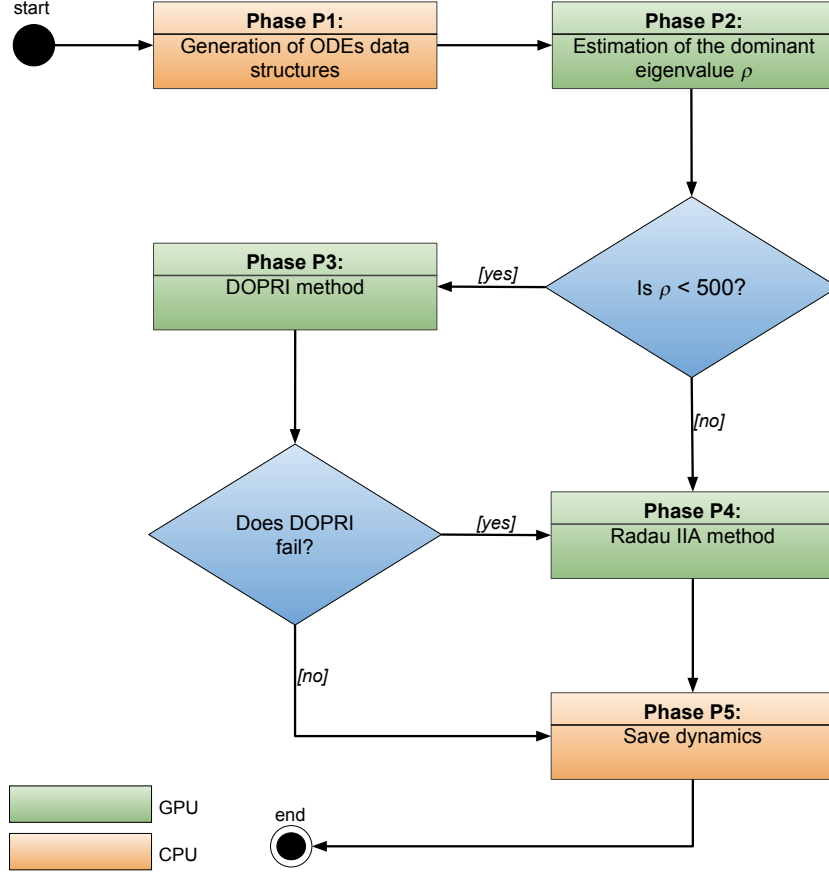


Figure 3: Simplified scheme of FiCoS workflow. During phase P_1 all the data structures used to encode the system of ODEs are generated. In phase P_2 each thread ϕ , with $\phi = 0, \dots, \Phi - 1$ estimates the dominant eigenvalue ρ related to the simulation ϕ , using the ϕ -th parameterization and the ϕ -th set of initial concentrations. All threads whose dominant eigenvalue ρ is less than 500 are added in the set \mathcal{L}_{DOPRI} , which contains the threads that use DOPRI as integration algorithm (phase P_3). If some simulations using DOPRI fail, those threads are added to the set \mathcal{L}_{RADAU} , which contains the threads that use Radau IIA as integration algorithm (phase P_4). As soon as the phases P_3 and P_4 are completed, the output data (i.e., the concentration values of molecular species sampled at fixed time instants) are transferred to the host that writes them in output files (phase P_5).

Phase P_1 . It is executed by the host and implements the generation of the data structures encoding the system of ODEs. For further details, see the Data structures and CUDA memory usage Section.

Phase P_2 . This phase implements the estimation of the dominant eigenvalue ρ of the Jacobian matrix \mathbf{J} by means of the norm bound of \mathbf{J} itself. As norm bound, we used the so-called maximum absolute row sum norm, defined as:

$$\|\mathbf{J}\|_{\infty} = \max_{1 \leq i \leq N} \sum_{j=1}^N |J_{ij}|. \quad (4)$$

Note that the dominant eigenvalue ρ is equal to the spectral radius, which is defined as the largest absolute value of the eigenvalues of \mathbf{J} .

Since the value ρ strictly depends on both the initial concentrations and the kinetic parameter values, ρ is calculated for each simulation ϕ , with $\phi = 0, \dots, \Phi - 1$, which must be performed. This phase is implemented using 2 different CUDA kernels:

- kernel K_1 : each thread ϕ , with $\phi = 0, \dots, \Phi - 1$, recalls the kernel K_{1a} to evaluate the Jacobian matrix \mathbf{J} using the ϕ -th parameterization and ϕ -th set of initial concentrations as state of the

system \mathbf{X} . As soon as \mathbf{J} is calculated, thread ϕ applies Eq. 4 to estimate the dominant eigenvalue ρ associated with the ϕ -th parameterization and ϕ -th set of initial concentrations;

- kernel K_{1_a} : each thread j , with $j = 0, \dots, N - 1$, calculates the values of the j -th row of \mathbf{J} using the state of the system \mathbf{X} and parameterization given as input.

Note that each thread ϕ , with $\phi = 0, \dots, \Phi - 1$, exploits the DP to call the kernel K_{1_a} , by launching a novel grid of threads.

Phase P_3 . It implements the DOPRI5 method [2, 3, 4], an explicit Runge-Kutta integration algorithm of order 5 with variable step-size and stiffness control, used by the threads whose dominant eigenvalue ρ is less than 500 to solve the system of ODEs. As a matter of fact, the Φ threads are partitioned in two different sets: \mathcal{L}_{DOPRI} that contains the threads whose dominant eigenvalue ρ is less than 500; \mathcal{L}_{RADAU} that contains the remaining threads. If the DOPRI5 method fails in solving the system of ODEs characterized by a parameterization $\phi = 0, \dots, \Phi$ and a set of initial concentrations, the ϕ -th thread is moved into the set \mathcal{L}_{RADAU} . The DOPRI5 method relies on the following 12 kernels:

- kernel K_2 : it is the main kernel implementing the DOPRI5 method. It is executed by each thread $d \in \mathcal{L}_{DOPRI}$, which uses its own parameterization as well as its own set of initial concentrations;
- kernel K_{2_a} : each thread j , with $j = 0, \dots, N - 1$, evaluates the j -th ODE using the state of the system \mathbf{X} and parameterization given as input;
- kernel K_{2_b} : it exploits N threads to update the vectors used to calculate the initial step-size of the DOPRI5 method;
- kernels $K_{2_c} - K_{2_i}$: each kernel uses N threads to update the vectors required by the DOPRI5 method to estimate the next step-size, using the Butcher tableau shown in Table 2 (left side);
- kernels K_{2_j} and K_{2_k} : in each kernel, N threads are launched to update the vectors involved in the spline approximation of the ODEs.

It is worth noting that each thread $d \in \mathcal{L}_{DOPRI}$ exploits the DP to call the kernels $K_{2_a} - K_{2_k}$, by launching a novel grid of threads. Since data transfers between the CPU and the GPU are very time consuming, all the temporary results computed by FiCoS during this phase are stored on the global memory of the GPU.

Phase P_4 . This phase implements the RADAU5 method [5, 6], an implicit Runge-Kutta integration algorithm of order 5 with variable step-size. It is executed after the phase P_3 , since the execution of some thread in that phase might fail. The RADAU5 method is based on the following 22 kernels:

- kernel K_3 : it is the main kernel implementing the RADAU5 method. It is executed by each thread $r \in \mathcal{L}_{RADAU}$, which uses its own parameterization as well as its own set of initial concentrations;
- kernel K_{1_a} : each thread j , with $j = 0, \dots, N - 1$, calculates the values of the j -th row of \mathbf{J} using the state of the system \mathbf{X} and the parameterization given as input;
- kernel K_{2_a} : each thread j , with $j = 0, \dots, N - 1$, evaluates the j -th ODE using the state of the system \mathbf{X} and parameterization given as input;
- kernel K_{3_a} : it uses N threads to reinitialize the vectors involved during the Newton—Raphson method [9];
- kernels K_{3_b} and K_{3_c} : each kernel takes advantage of N threads to update the matrices—which are linearized—used during the linear system resolutions required by RADAU5 method;
- kernels K_{3_d} and K_{3_i} : in each kernel, N threads are launched to update the vectors used before and after the linear system resolutions required by RADAU5 method, relying on the Butcher tableau depicted in Table 2 (right side);

- kernels K_{3_m} and K_{3_o} : these 3 kernels are used during the error estimation phase in which the new step-size is computed. Also in this case, each kernel takes advantage of N threads to update the vectors required to calculate the new step-size based on the estimated error;
- kernels K_{3_p} and K_{3_s} : in each kernel, N threads are used to update the vectors involved in the spline approximation of the ODEs.

We highlight that both the matrix decompositions and the linear system resolutions were implemented exploiting the cuBLAS library [12]. Each thread $r \in \mathcal{L}_{RADAU}$ takes advantage of the DP to call the kernels K_{1_a} , K_{2_a} , $K_{3_a} - K_{3_s}$ launching a novel grid of threads. Since data transfers between the CPU and the GPU are very time consuming, all the temporary results computed by FiCoS during this phase are stored on the global memory of the GPU.

Phase P_5 . During this phase, for each simulation, the dynamics of the species to be sampled are written into output files. These output data (i.e., the concentration values of molecular species sampled at fixed time instants) are transferred to the host as soon as the phases P_3 and P_4 are completed.

4 Results

The results of the tests proposed in this section aim at showing the simulation accuracy of FiCoS, as well as the achieved speed-up, with respect to Livermore Solver of Ordinary Differential Equations (LSODA) [13], Variable-coefficient ODE (VODE) [14], cupSODA [15], and LASSIE (LArge-Scale Simulator) [16]. We leveraged the LSODA and VODE implementations provided by the SciPy scientific library (v.0.18.1) [17], and we used Python (v.2.7.12) and the NumPy library (v.1.11.2). cupSODA and LASSIE were downloaded from the github repositories <https://github.com/aresio/LASSIE> and <https://github.com/aresio/cupSODA>. All tests were performed on a workstation equipped with a CPU Intel Core i7-2600 CPU (clock 3.4 GHz) and 8 GB of RAM, running on Ubuntu 16.04 LTS. The GPU used in the tests was a Nvidia GeForce GTX Titan X (3072 cores, clock 1.075 GHz, RAM 12 GB), CUDA toolkit version 8 (driver 387.26).

4.1 Simulation accuracy of FiCoS

The simulation accuracy of FiCoS was evaluated by using a CPU implementation of the numerical integrators ported on the GPU.

First, we compared the dynamics of the molecular species *EIF4EBP* and *AMBRA* of the RBM of the Autophagy/Translation switch based on the mutual inhibition of MTORC1 and ULK1 [18], obtained by running LSODA, VODE, and FiCoS. The dynamics of this RBM was simulated by setting the initial value of *AMPK* species to 90000, which allows for obtaining an oscillating behavior, thus making the system of ODEs stiff. We used the following settings for all simulators: (i) absolute tolerance $\varepsilon_a = 10^{-12}$; (ii) relative tolerance $\varepsilon_r = 10^{-6}$ and (iii) maximum number of allowed steps equal to 10^4 . Fig. 4 shows that the dynamics obtained by FiCoS perfectly overlap those provided by LSODA and VODE, proving the correctness of the implementation of FiCoS as well as its accuracy.

Second, to assess the robustness of FiCoS with respect to the numerical integrators settings (i.e., absolute ε_a and relative ε_r tolerances), we run a batch of tests in which we changed one of these parameters at a time. The same tests were executed with LSODA and VODE, for a fair comparison. We evaluated the effect of these settings on both the quality of the output dynamics and the running time, considering the RBM of the Ras/cAMP/PKA signal transduction pathway in yeast [19, 20], which consists in 39 reactions and 33 molecular species, and it is characterized by an oscillatory behavior leading to the stiffness phenomenon.

The accuracy of the simulations outcome of LSODA, VODE, and FiCoS were assessed by comparing the dynamics of the molecular species cAMP simulated with different settings (see Table 3), against a reference dynamics obtained by running LSODA with the following settings: $\varepsilon_a = 10^{-12}$, $\varepsilon_r = 10^{-6}$ and maximum number of allowed steps equal to 10^4 . Figs. 5, 6, 7 show the results of this tests.

We observe that, in the case of FiCoS and VODE, all the dynamics are perfectly overlapped, as the tolerance values have no relevant effect on the quality of the output dynamics. Conversely, the dynamics obtained with LSODA changes with different tolerance settings. In the case of $\varepsilon_r = 10^{-2}$ and

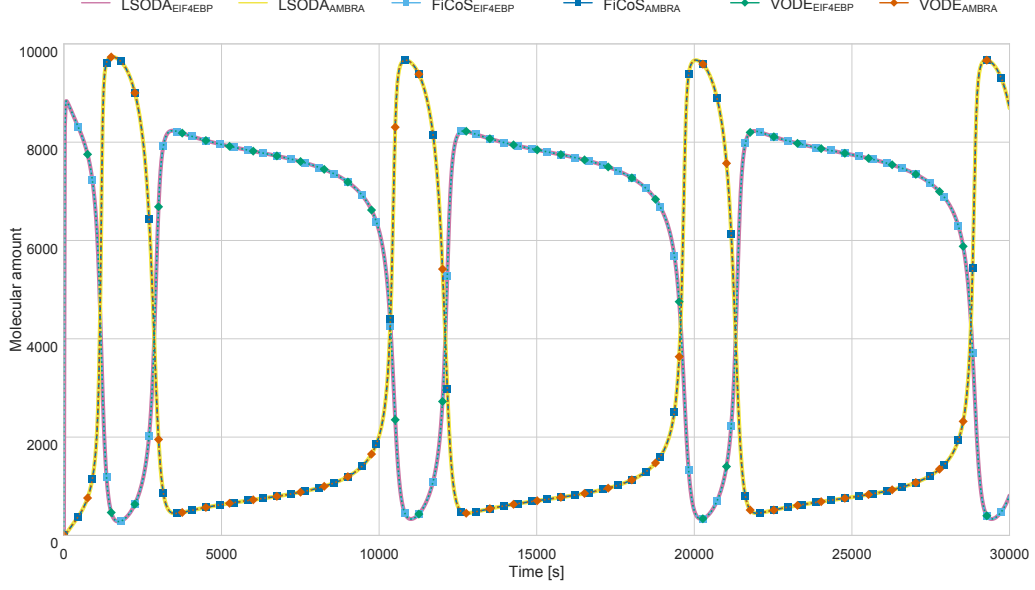


Figure 4: Comparison of the dynamics of the molecular species *EIF4EBP* and *AMBRA* of the model presented in [18], obtained by running FiCoS, LSODA and VODE, by using $\varepsilon_a = 10^{-12}$ and $\varepsilon_r = 10^{-6}$.

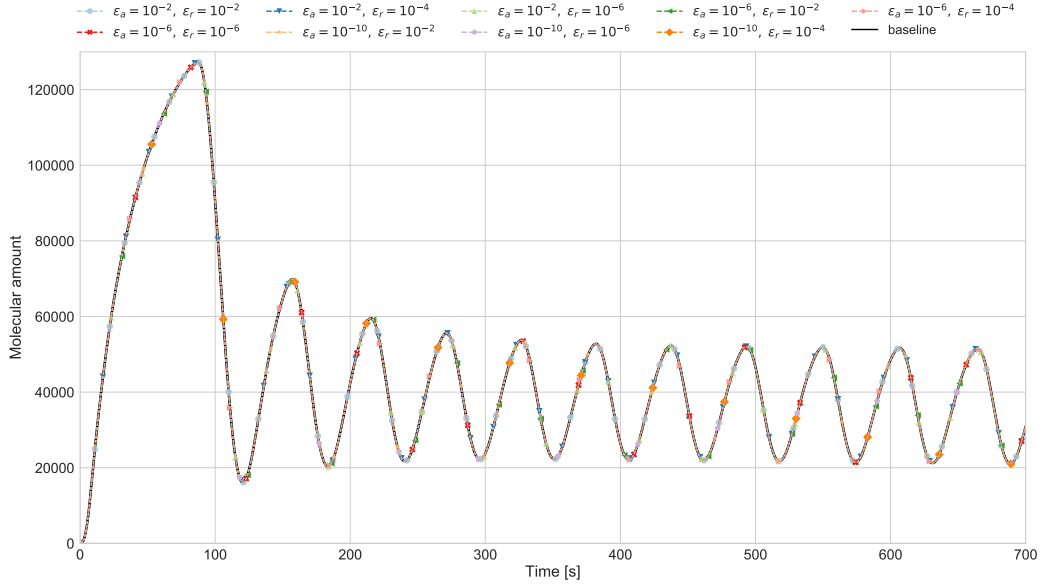


Figure 5: Comparison of the dynamic of the molecular species cAMP of the model of the RAS/cAMP/PKA signaling pathway in yeast, obtained by FiCoS varying $\varepsilon_a \in 10^{-2}, 10^{-6}, 10^{-10}$ and $\varepsilon_r \in 10^{-2}, 10^{-4}, 10^{-6}$.

$\varepsilon_a \in 10^{-2}, 10^{-6}$, the output dynamics do not perfectly overlap the reference dynamics (Fig. 7). This is probably caused by the automatically switching between the explicit and implicit families of integration methods. Since the explicit methods are not accurate for stiff systems, they are not able to properly integrate this ODE system, thus reducing the accuracy of the simulated dynamics.

Table 3 summarizes the running time required by LSODA, VODE and FiCoS to solve the systems of ODEs related to the RAS/cAMP/PKA signaling pathway in yeast, for each tested combination of (ε_a) and (ε_r) values. We observe that, while the running time of VODE remains almost constant with different tolerances values, FiCoS and LSODA performance can be dramatically affected by this choice,

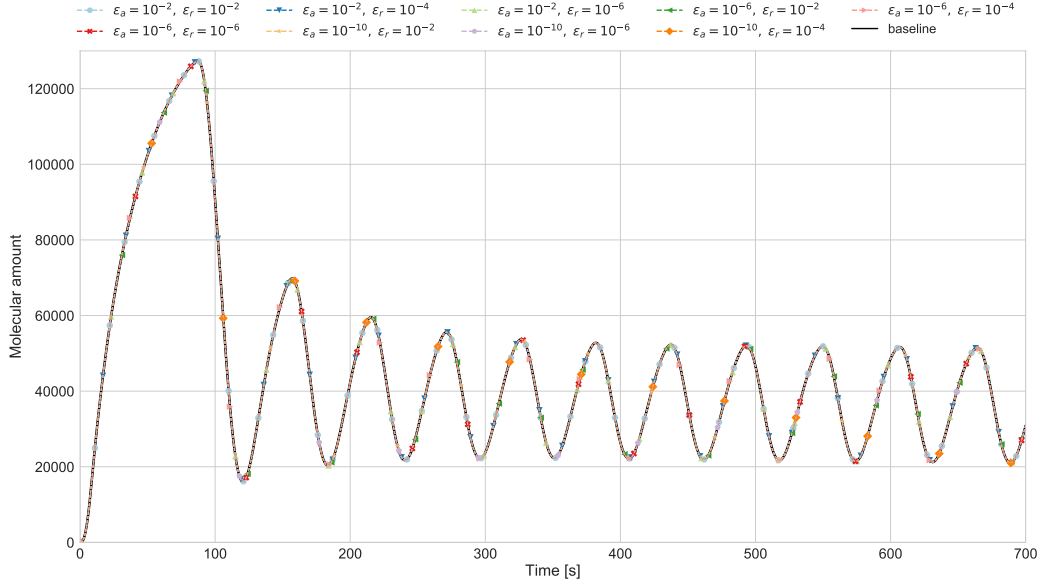


Figure 6: Comparison of the dynamic of the molecular species cAMP of the model of the RAS/cAMP/PKA signaling pathway in yeast, obtained by VODE varying $\varepsilon_a \in 10^{-2}, 10^{-6}, 10^{-10}$ and $\varepsilon_r \in 10^{-2}, 10^{-4}, 10^{-6}$.

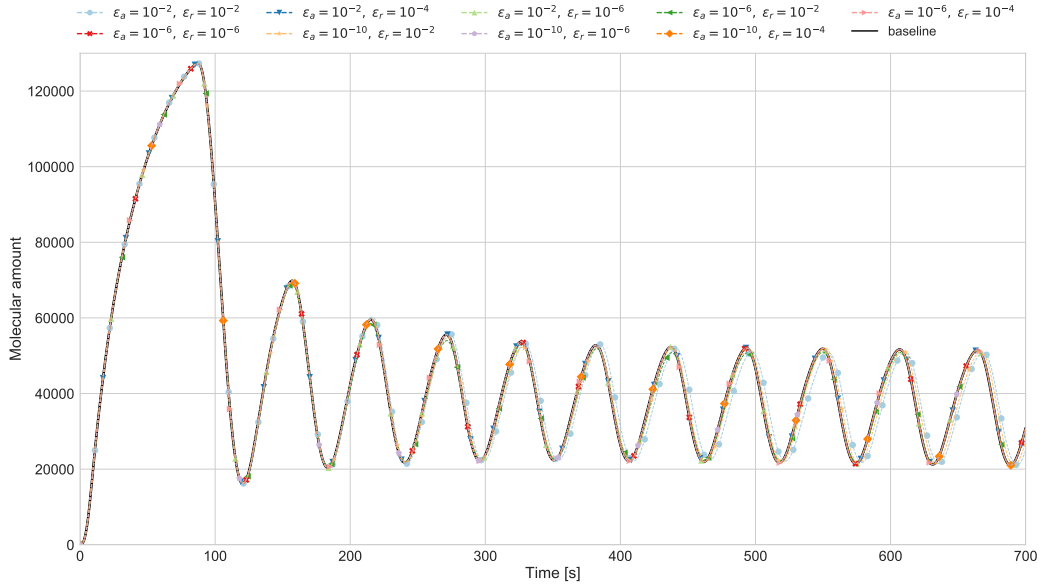


Figure 7: Comparison of the dynamic of the molecular species cAMP of the model of the RAS/cAMP/PKA signaling pathway in yeast, obtained by LSODA varying $\varepsilon_a \in 10^{-2}, 10^{-6}, 10^{-10}$ and $\varepsilon_r \in 10^{-2}, 10^{-4}, 10^{-6}$.

as the running times increase up to three times with some tolerance values combinations. To better investigate this aspect, we defined another batch of tests in which we varied the tolerances values and analyze the running time required to simulate the Autophagy/Translation switch based on the mutual inhibition of MTORC1 and ULK1 model, which is characterized by 173 molecular species and 6581 reactions. Table 4 shows that the running time of VODE double from the combination $\varepsilon_a = \varepsilon_r = 10^{-2}$ to the combination $\varepsilon_a = 10^{-10}$ and $\varepsilon_r = 10^{-6}$, while the running time of LSODA and FiCoS triple. We also observe that the relative tolerance has a higher impact on the running time with respect to the

Table 3: Running time required by LSODA, VODE, and FiCoS to simulate the RBM of the RAS/cAMP/PKA signaling pathway in yeast, with different settings of absolute (ε_a) and relative (ε_r) tolerance.

Settings		Running time [s]		
ε_a	ε_r	LSODA	VODE	FiCoS
10^{-2}	10^{-2}	0.255	2.516	0.030
10^{-2}	10^{-4}	0.306	2.538	0.041
10^{-2}	10^{-6}	0.381	2.419	0.030
10^{-6}	10^{-2}	0.339	2.508	0.043
10^{-6}	10^{-4}	0.407	2.465	0.059
10^{-6}	10^{-6}	0.579	2.419	0.081
10^{-10}	10^{-2}	0.324	2.533	0.046
10^{-10}	10^{-4}	0.373	2.481	0.034
10^{-10}	10^{-6}	0.601	2.501	0.052

Table 4: Running time required by LSODA, VODE, and FiCoS to simulate the Autophagy/Translation switch based on the mutual inhibition of MTORC1 and ULK1 model, with different settings of absolute (ε_a) and relative (ε_r) tolerance.

Settings		Running time [s]		
ε_a	ε_r	LSODA	VODE	FiCoS
10^{-2}	10^{-2}	12.599	25.922	2.398
10^{-2}	10^{-4}	16.348	27.101	2.791
10^{-2}	10^{-6}	19.722	29.449	3.311
10^{-6}	10^{-2}	13.261	31.077	2.661
10^{-6}	10^{-4}	19.433	32.822	3.717
10^{-6}	10^{-6}	33.544	43.372	5.655
10^{-10}	10^{-2}	13.747	32.503	2.774
10^{-10}	10^{-4}	21.219	33.860	4.329
10^{-10}	10^{-6}	37.02	56.687	6.952

absolute tolerance: in the case of FiCoS, with $\varepsilon_a = 10^{-10}$, the running time triple from $\varepsilon_r = 10^{-2}$ to $\varepsilon_r = 10^{-6}$.

Finally, this tests show that the CPU implementation of FiCoS is faster than LSODA and VODE, with up to $84\times$ speed-up, achieving in some cases more accurate dynamics.

4.2 Computational performance

Starting from the results presented in the previous section, we present here the investigation of the computational performance of the GPU-powered implementation of FiCoS. The following simulations were executed with absolute tolerance $\varepsilon_a = 10^{-12}$, relative tolerance $\varepsilon_r = 10^{-6}$ and maximum number of allowed steps equal to 10^4 , which represent settings widely used in the literature as well as in well-known simulation software like COPASI [21]. We also highlight that in every simulation we save the dynamics of all species included in the RBMs. For each RBM, we performed an increasing number of simulations (up to 2048) and calculated both the integration time and the simulation time. The integration time indicates the running time spent by the numerical integration algorithms to solve the system of ODEs, while the simulation time is the overall running time required to perform a simulation, including the I/O operations (i.e., reading and writing operations). By using these data, we calculated the speed-up achieved by FiCoS with respect to the CPU-based ODE solver LSODA[13] and VODE[14], and the GPU-powered algorithms cupSODA [15] and LASSIE [16].

We carried out different batches of simulations using a set of “symmetric” synthetic RBMs of increasing size, ranging from 64 to 800 species and reactions, to evaluate the impact of the models size on the computational performance, and a set of “asymmetric” synthetic RBMs of increasing size, ranging from

21 species (reactions) and 64 reactions (species) to 267 species (reactions) and 800 reactions (species). The asymmetric RBMs were used to evaluate how much either the number of species or the number of reactions affect the performance of FiCoS.

Symmetric models. Figs. 8-11 present the results obtained with the batch of tests on symmetric RBMs. The panels in the figures report the speed-up values achieved by FiCoS, calculated considering the entire simulation time (a) and integration time only (b), with respect to LSODA, VODE, LASSIE, and cupSODA.

FiCoS provides a relevant reduction of the entire simulation time, achieving a $360\times$ speed-up against LSODA (Fig. 8a), and a $487\times$ speed-up against VODE (Fig. 9a) in the case of 2048 simulations of the 800×800 RBM. Note that the maximum speed-up value with respect to LSODA ($366\times$) is reached in the case of 1024 simulations of the 800×800 RBM. We conjecture that for RBMs of larger size, the speed-up could be even higher with respect to a CPU-bound execution; however, we did not perform such tests due to the excessive memory requirements of LSODA and VODE implementations.

Since FiCoS was designed to exploit both a fine- and a coarse-grained strategy, the intra-GPU communication overhead—caused by the fine-grain kernels—makes the simulation to less efficient in the case of small size RBMs. Moreover, FiCoS performance can be hampered when a few simulations are executed. For instance, in the case of a single simulation of a RBM of size up to 256×256 , FiCoS is slower than both LSODA (Fig. 8a) and VODE (Fig. 9a). Anyway, while for a single simulation of small and medium RBMs a CPU-bound simulation should be preferred, in all other cases FiCoS represents the best solution.

Considering the performance achieved in the case of high numbers of simulations, we observed that an “excessive” parallelization might cause the saturation of the computing resources of the GPU. This is due to the computing resources required by the fine-grained kernels, exploited to parallelize the resolution of the system of ODEs, which increase along with the RBM size. As a matter of fact, the blocks of threads generated using the DP saturate the GPU resources, resulting in a decrease of the speed-up. In our tests, this phenomenon occurs for RBMs with more than 512 species and reactions, in the case of 2048 simulations.

Restricting the analysis of the performance of FiCoS to the integration time only, the portion of the running time spent to reading/writing operations of the input/output files, is excluded from the computation of the speed-up values. The speed-up achieved by the integration method implemented in FiCoS is smaller than the speed-up of the entire simulation time in the case of LSODA, with a speed-up up to $79\times$, while the speed-up with respect to VODE increases, up to $855\times$ (see Figs. 8b and 9b, respectively). Overall, these results are consistent with the previous findings: FiCoS is faster than LSODA and VODE in all cases, except for the single simulation of the RBMs, and becomes less effective in the scarcity of GPU resources, e.g., when more than 2000 parallel simulations are executed.

Regarding the comparison of FiCoS against LASSIE, since the latter exploits only a single parallelization strategy, as shown in Fig. 10a, it is faster than FiCoS in the case of a single simulation (except for the RBM with size 512×512). Regarding the integration time, FiCoS always outperformed LASSIE, achieving a maximum speed-up equal to $5.57\times$ (Fig. 10a). Considering tasks with multiple simulations, FiCoS resulted always faster than LASSIE; the highest speed-up is achieved in the case of 2048 simulations of the RBM with size 64×64 , in which FiCoS is $298\times$ faster than LASSIE for the simulation time (Fig. 10a), and $760\times$ faster for the integration time (Fig. 10b).

We finally compared the performance of FiCoS against cupSODA, a GPU-powered simulator designed for small RBMs. cupSODA resulted faster with respect FiCoS in the case of a single simulation when less than 512 species and reactions are taken into account (see Fig. 11a). In the case of the smallest RBM considered in these tests (i.e., 64 species and reactions), FiCoS is more efficient only when more than 128 simulations are executed. In all remaining cases, FiCoS is the best choice, achieving a speed-up up to $7.35\times$. Regarding the integration time, FiCoS always outperformed cupSODA, reaching the maximum speed-up ($17\times$) in the case of 128 simulations of the RBM with 800 species and reactions (see Fig. 11b). Note that, due to the required amount of memory, cupSODA failed to perform 2048 simulations of the RBMs with 640 and 800 species and reactions.

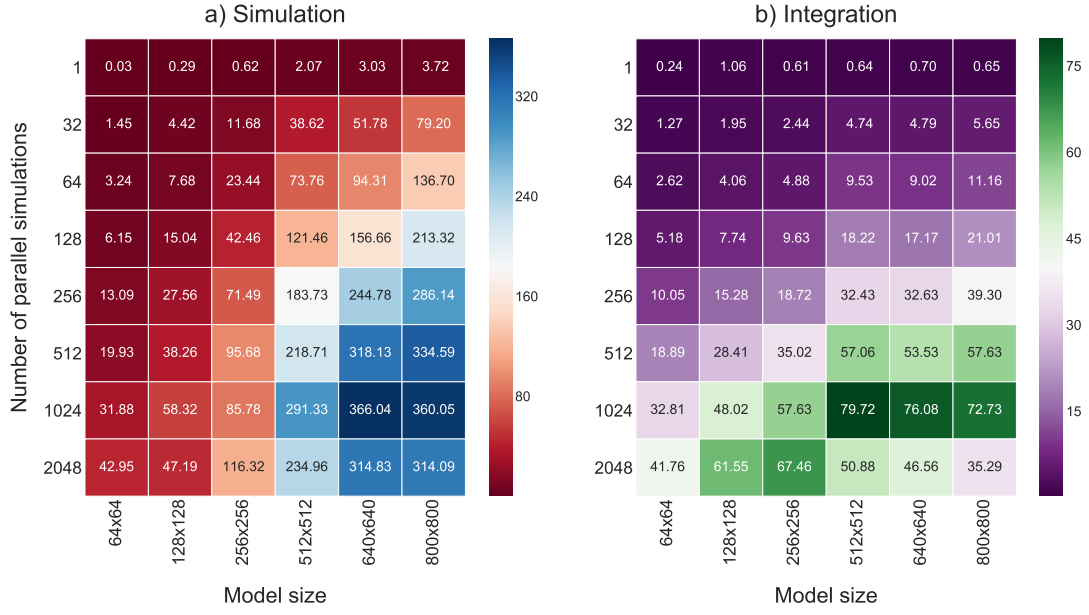


Figure 8: Speed-up provided by FiCoS with respect to LSODA[13] in the case of symmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and the number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively.

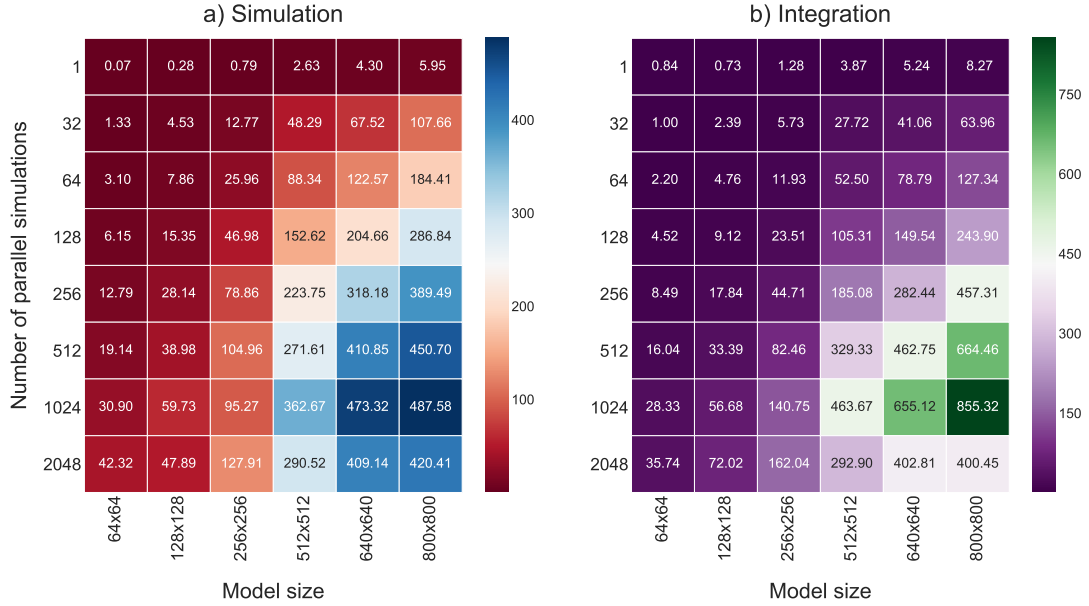


Figure 9: Speed-up provided by FiCoS with respect to VODE[14] in the case of symmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and the number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively.

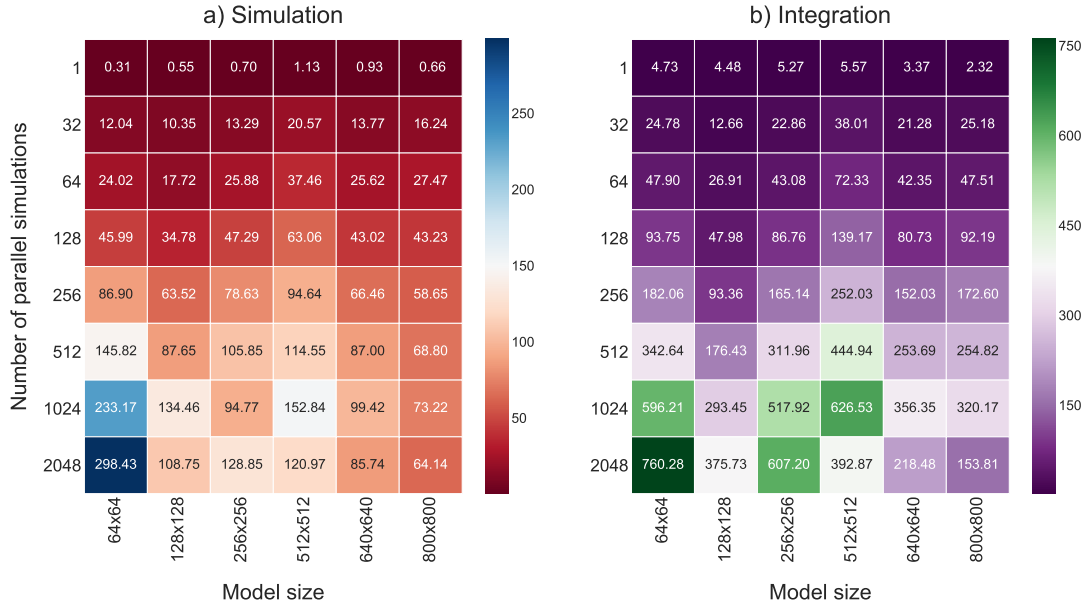


Figure 10: Speed-up provided by FiCoS with respect to LASSIE[16] in the case of symmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and the number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time the numerical integration time, respectively.

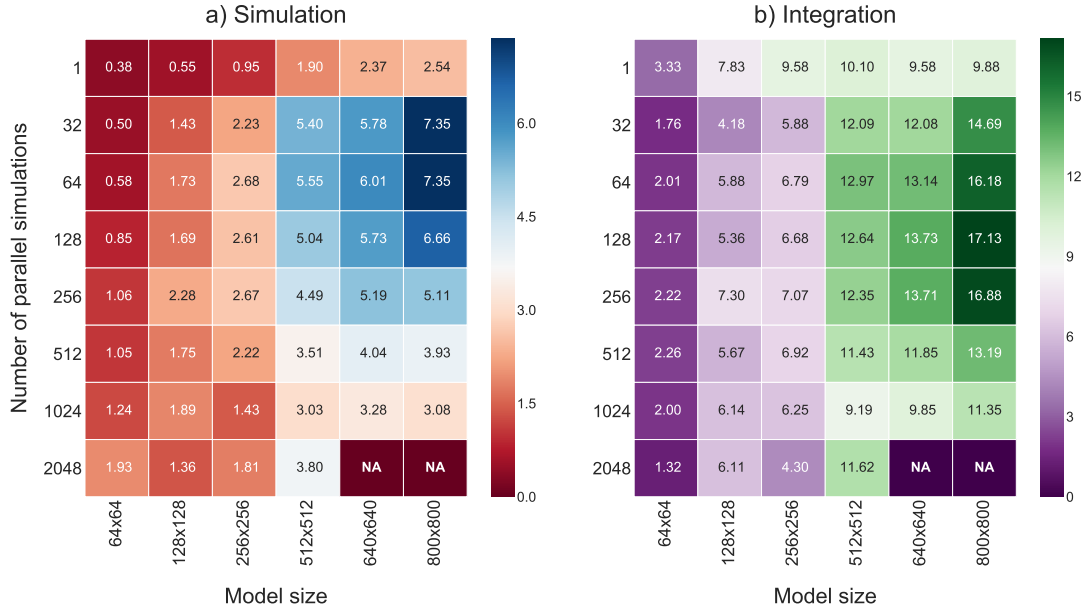


Figure 11: Speed-up provided by FiCoS with respect to cupSODA[15] in the case of symmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and the number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively. NA values denote that cupSODA failed to perform the simulations due to the required amount of memory.

Asymmetric models. Figs. 12-19 present the results obtained with the batch of tests on asymmetric RBMs. The panels in these figures report the speed-up values achieved by FiCoS, calculated considering the entire simulation time (a) and the integration time only (b), with respect to LSODA, VODE, LASSIE, and cupSODA. In particular, to investigate how the number of species and reactions could affect the performance of the simulators (especially those exploiting the GPGPU computing), we generated a first set of asymmetric RBMs, in which the number of species is three times the number of reactions, and a second set having RBMs with opposite characteristics. These tests aim at analyzing the capabilities of FiCoS, which realizes both a fine- and coarse-grained parallelization strategy. Indeed, since each ODE corresponds to a different species, the higher the number of the species the higher the parallelization that can be obtained by exploiting a fine-grained strategy.

Conversely, since the number of reactions is roughly related to the length of each ODE, the number of operations that must be performed by each thread increases along with the number of reactions; therefore, the coarse-grained parallelization strategy might help mitigating the running time required.

Considering the first set of asymmetric RBMs, FiCoS was able to achieve speed-up values up to $413\times$ and up to $508\times$ with respect to LSODA and VODE, respectively, for what concerns the simulation time (Figs. 12a and 13a). Although these values are higher than those obtained in the case of symmetric RBMs (Figs. 8a and 9a), regarding the integration time, the speed-ups achieved by FiCoS are lower. LASSIE outperformed FiCoS only when a single simulation is performed. Considering the simulation time, the maximum speed-up is $255\times$, while in the case of the integration time is $278\times$ (Figs. 14 and 10). Finally, comparing FiCoS with cupSODA, the maximum speed-ups are halved with respect to case of symmetric RBMs (see Figs. 15 and 11). Moreover, cupSODA is faster than FiCoS for tasks involving up to 256 simulations of RBMs having size up to 256×85 .

Considering the second set of asymmetric models, as shown in Figs. 16a and 17a, the speed-up of FiCoS is up to $252\times$ and $277\times$ with respect to LSODA and VODE, respectively, for the overall simulation time. Regarding the integration time, FiCoS achieved a speed-up up to $147\times$ and up to $234\times$ with respect to LSODA and VODE, respectively (Fig. 16b and Fig. 17b). Except for the single simulation of the RBM having size 21×64 , LASSIE was always slower than FiCoS, the latter being capable of achieving speed-ups values up to $1100\times$ considering the simulation time and $3666\times$ considering the integration time (see Fig. 18). These tests evidenced the limitations of LASSIE, which implements only a fine-grained parallelization strategy, while FiCoS is capable of better exploiting the computing capabilities of the modern GPU. Finally, the comparison between FiCoS and cupSODA (Fig. 19) resulted in speed-up values up to $27\times$ for the simulation time, and up to $60\times$ for the integration time. cupSODA is more efficient than FiCoS only when simulating small-size RBMs; however, it failed to perform the simulations of RBMs of size 267×800 due to the required amount of memory.

Overall, these results confirmed that simulators exploiting only a fine- or a coarse-grained strategy are very inefficient in certain conditions, while the solution implemented in FiCoS allows for obtaining excellent performance in all computationally demanding tasks.

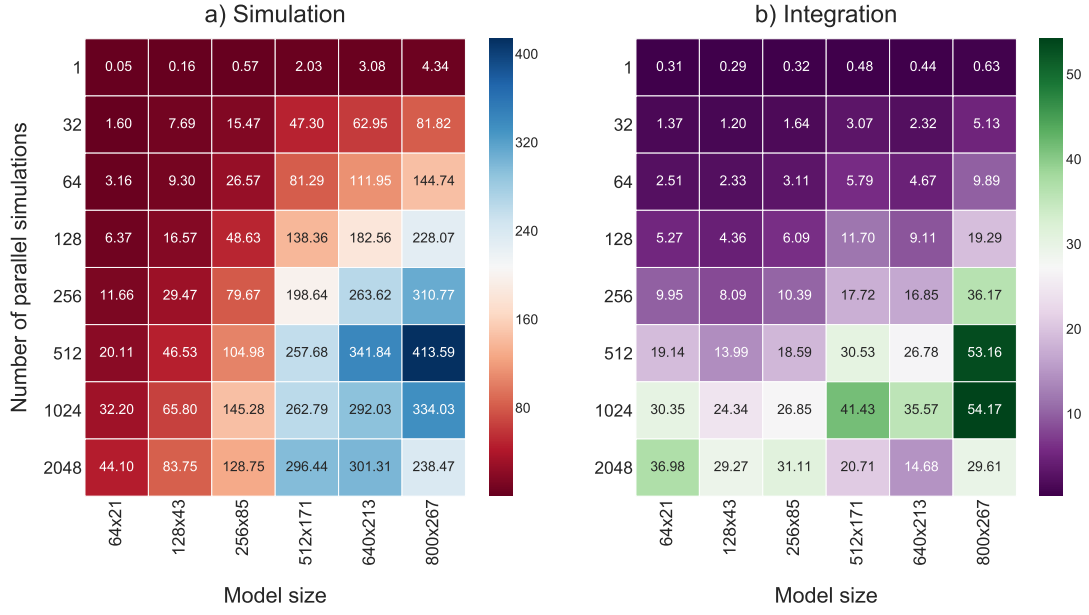


Figure 12: Speed-up provided by FiCoS with respect to LSODA[13] in the case of asymmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively.

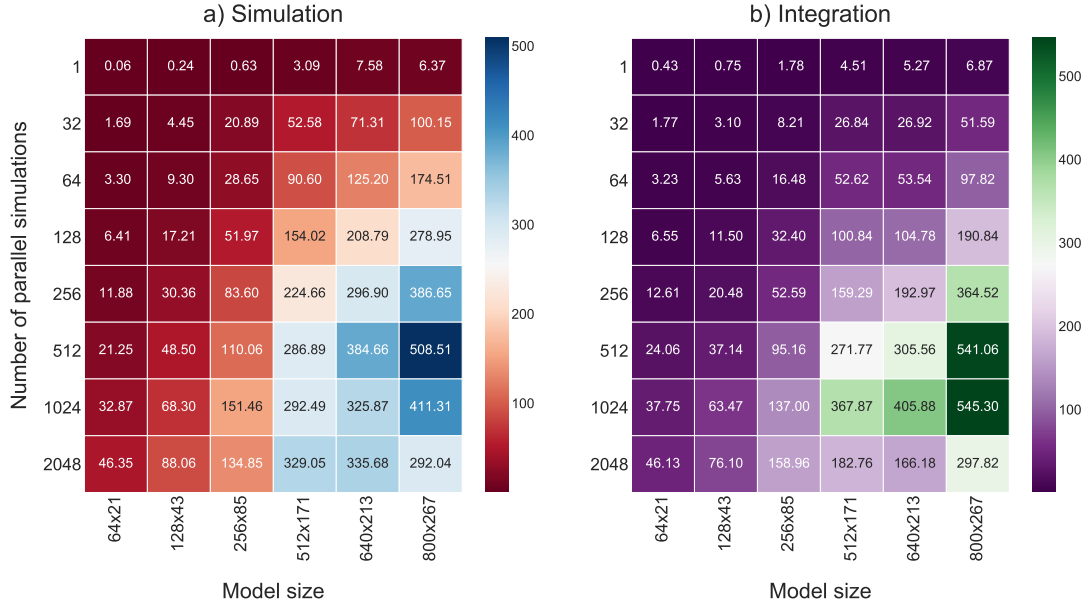


Figure 13: Speed-up provided by FiCoS with respect to VODE[14] in the case of asymmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively.

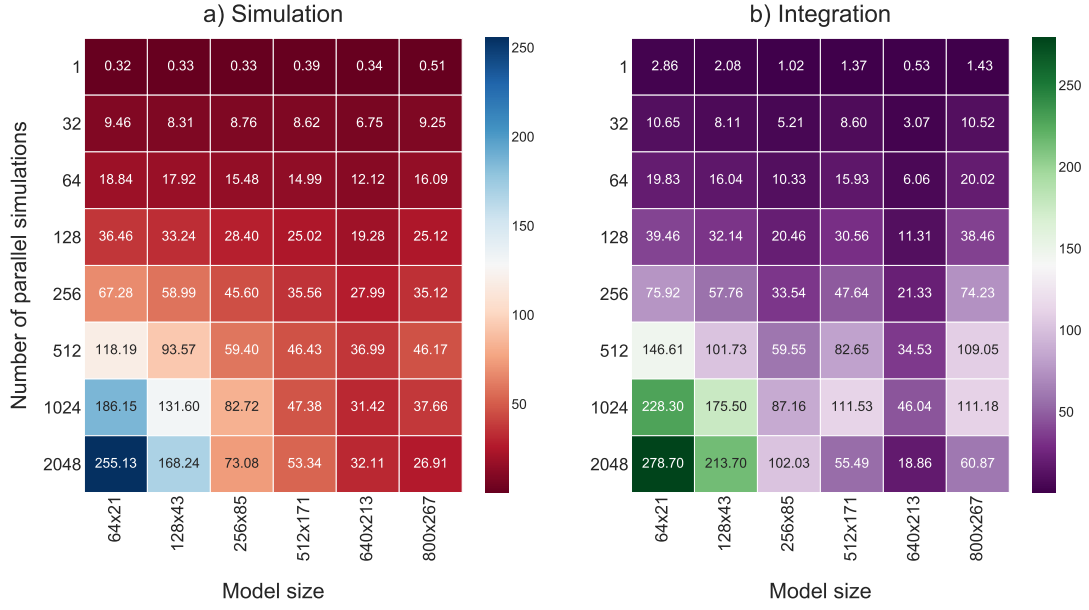


Figure 14: Speed-up provided by FiCoS with respect to LASSIE[16] in the case of asymmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively.

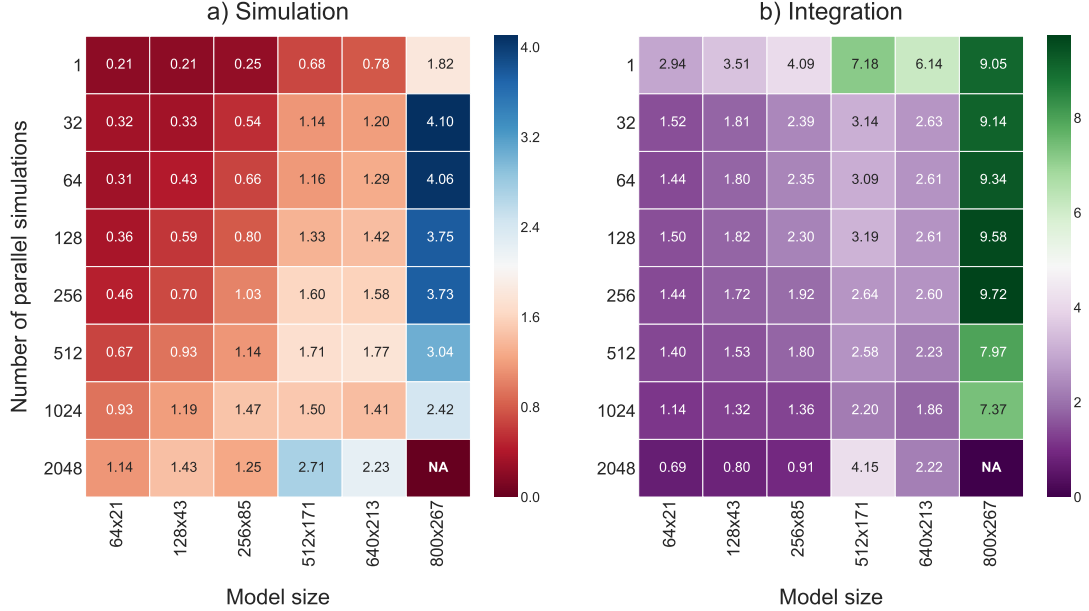


Figure 15: Speed-up provided by FiCoS with respect to cupSODA[15] in the case of asymmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively. NA values denote that cupSODA failed to perform the simulations due to the required amount of memory.

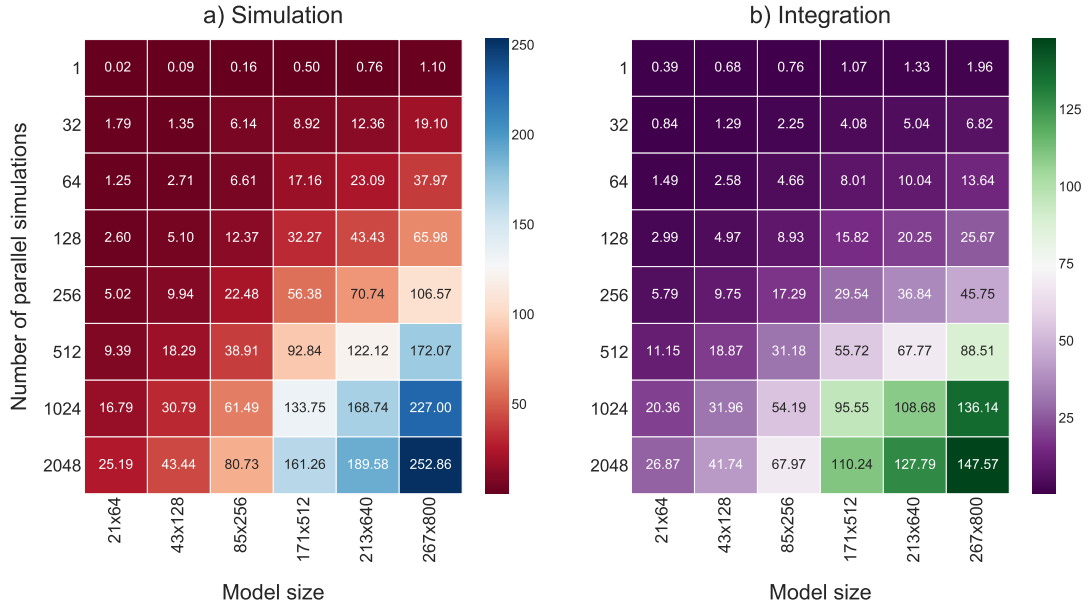


Figure 16: Speed-up provided by FiCoS with respect to LSODA[13] in the case of asymmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively.

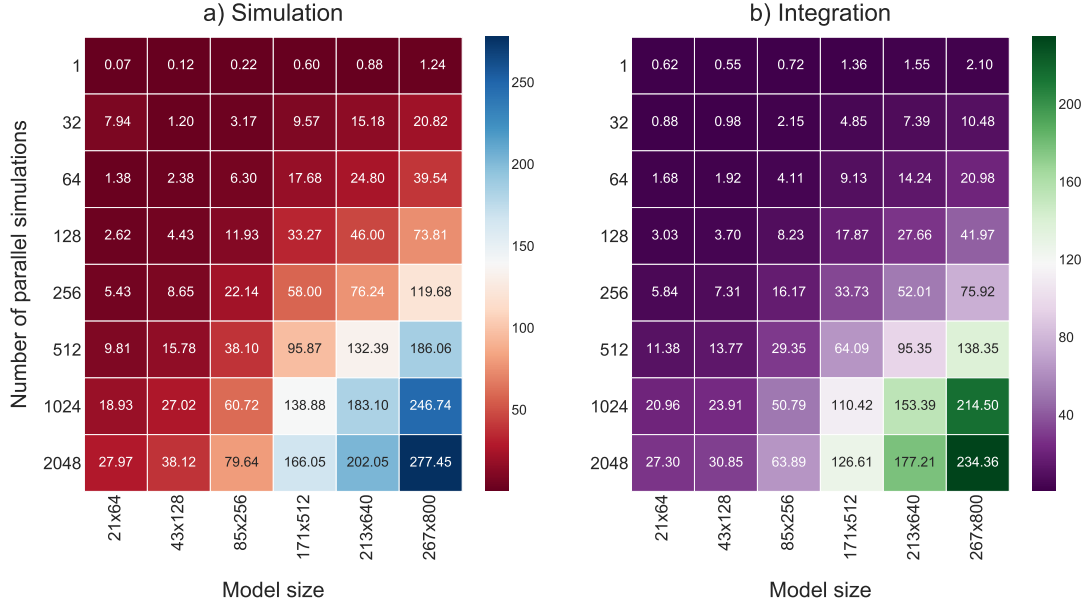


Figure 17: Speed-up provided by FiCoS with respect to VODE[14] in the case of asymmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively.

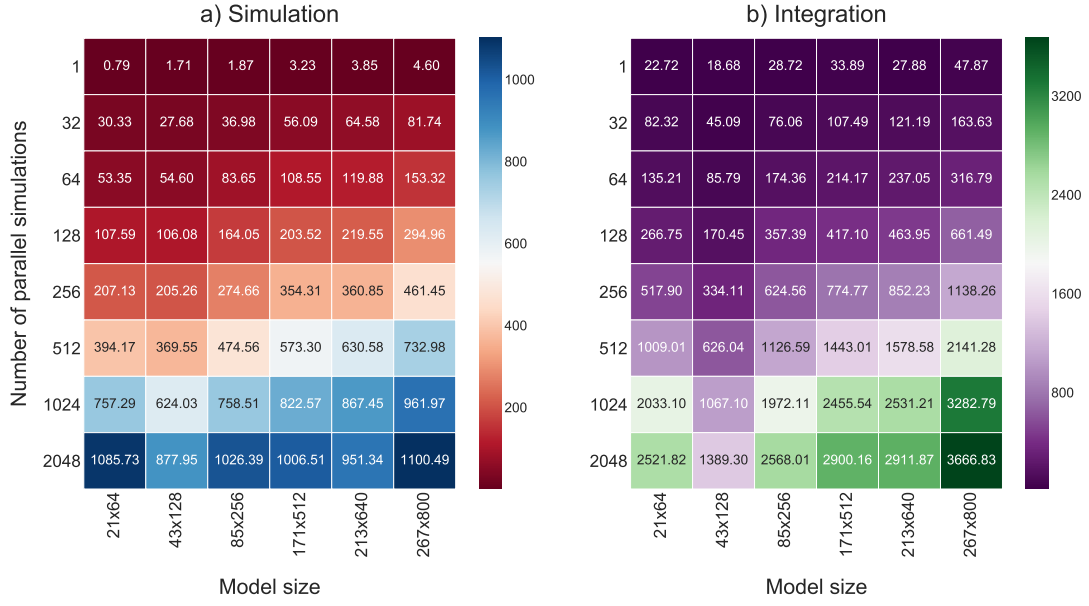


Figure 18: Speed-up provided by FiCoS with respect to LASSIE[16] in the case of asymmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively.

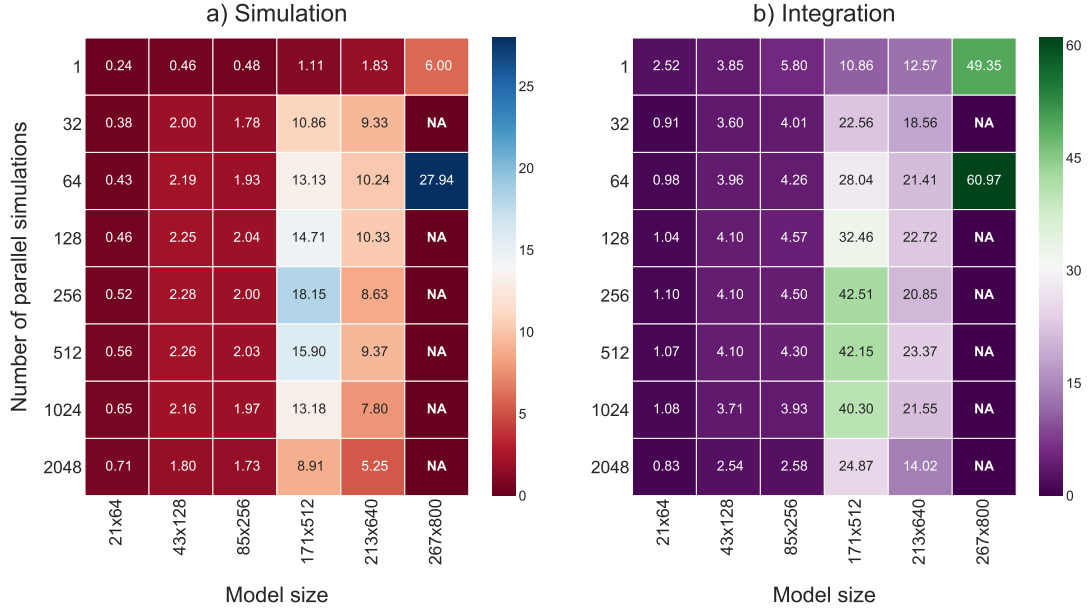


Figure 19: Speed-up provided by FiCoS with respect to cupSODA[15] in the case of asymmetric RBMs. The speed-up is analyzed by increasing both the size of the RBM (horizontal axis) and number of parallel simulations (vertical axis). Panel a) and b) show the speed-up calculated considering the entire simulation time and numerical integration time, respectively. NA values denote that cupSODA failed to perform the simulations due to the required amount of memory.

References

- [1] A. Prothero and A. Robinson. On the stability and accuracy of one-step methods for solving stiff systems of ordinary differential equations. *Math. Comp.*, 28(125):145–162, 1974.
- [2] J. R. Dormand and P. J. Prince. A family of embedded Runge-Kutta formulae. *J. Comput. Appl. Math.*, 6(1):19–26, 1980.
- [3] J. R. Dormand. *Numerical methods for differential equations: a computational approach*, volume 3. CRC Press, 1996.
- [4] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I*, volume 8. Springer-Verlag, Berlin, 2008.
- [5] E. Hairer and G. Wanner. *Solving ordinary differential equations II*, volume 14. Springer-Verlag, Berlin, 2002.
- [6] E. Hairer and G. Wanner. Stiff differential equations solved by Radau methods. *J. Comput. Appl. Math.*, 111(1-2):93–111, 1999.
- [7] L.F. Shampine. Some practical Runge-kutta formulas. *Math. Comp.*, 46(173):135–150, 1986.
- [8] W.H. Enright, K.R. Jackson, S.P. Nørsett, and P.G. Thomsen. Interpolants for Runge-Kutta formulas. *ACM Trans. Math. Softw.*, 12(3):193–218, 1986.
- [9] A. Ben-Israel. A Newton-Raphson method for the solution of systems of equations. *J. Math. Anal. Appl.*, 15(2):243–252, 1966.
- [10] R. H. Bartels and G. H. Golub. The simplex method of linear programming using LU decomposition. *Commun. ACM*, 12(5):266–268, 1969.
- [11] Nvidia. CUDA C Programming Guide, version 8.0, 2016.
- [12] Nvidia. cuBLAS library 8.0, 2015.
- [13] Linda Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM J. Sci. Stat. Comp.*, 4(1):136–148, 1983.
- [14] Peter N Brown, George D Byrne, and Alan C Hindmarsh. VODE: A variable-coefficient ODE solver. *SIAM J. Sci. Stat. Comp.*, 10(5):1038–1051, 1989.
- [15] M. S. Nobile, P. Cazzaniga, D. Besozzi, and G. Mauri. GPU-accelerated simulations of mass-action kinetics models with cupSODA. *J. Supercomput.*, 69(1):17–24, 2014.
- [16] A. Tangherloni, M. S. Nobile, D. Besozzi, G. Mauri, and P. Cazzaniga. LASSIE: simulating large-scale models of biochemical systems on GPUs. *BMC Bioinform.*, 18(1):246, 2017.
- [17] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001.
- [18] P. Szymańska, K. R. Martin, J. P. MacKeigan, W. S. Hlavacek, and T. Lipniacki. Computational analysis of an Autophagy/Translation switch based on mutual inhibition of MTORC1 and ULK1. *PLoS One*, 10(3):e0116550, 2015.
- [19] P. Cazzaniga, D. Pescini, D. Besozzi, G. Mauri, S. Colombo, and E. Martegani. Modeling and stochastic simulation of the ras/camp/pka pathway in the yeast *saccharomyces cerevisiae* evidences a key regulatory function for intracellular guanine nucleotides pools. *J. Biotechnol.*, 133(3):377–385, 2008.
- [20] D. Pescini, P. Cazzaniga, D. Besozzi, G. Mauri, L. Amigoni, S. Colombo, and E. Martegani. Simulation of the ras/camp/pka pathway in budding yeast highlights the establishment of stable oscillatory states. *Biotechnol. Adv.*, 30(1):99–107, 2012.
- [21] S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer. COPASI - a COMplex PATHway SIMulator. *Bioinformatics*, 22(24):3067–3074, 2006.